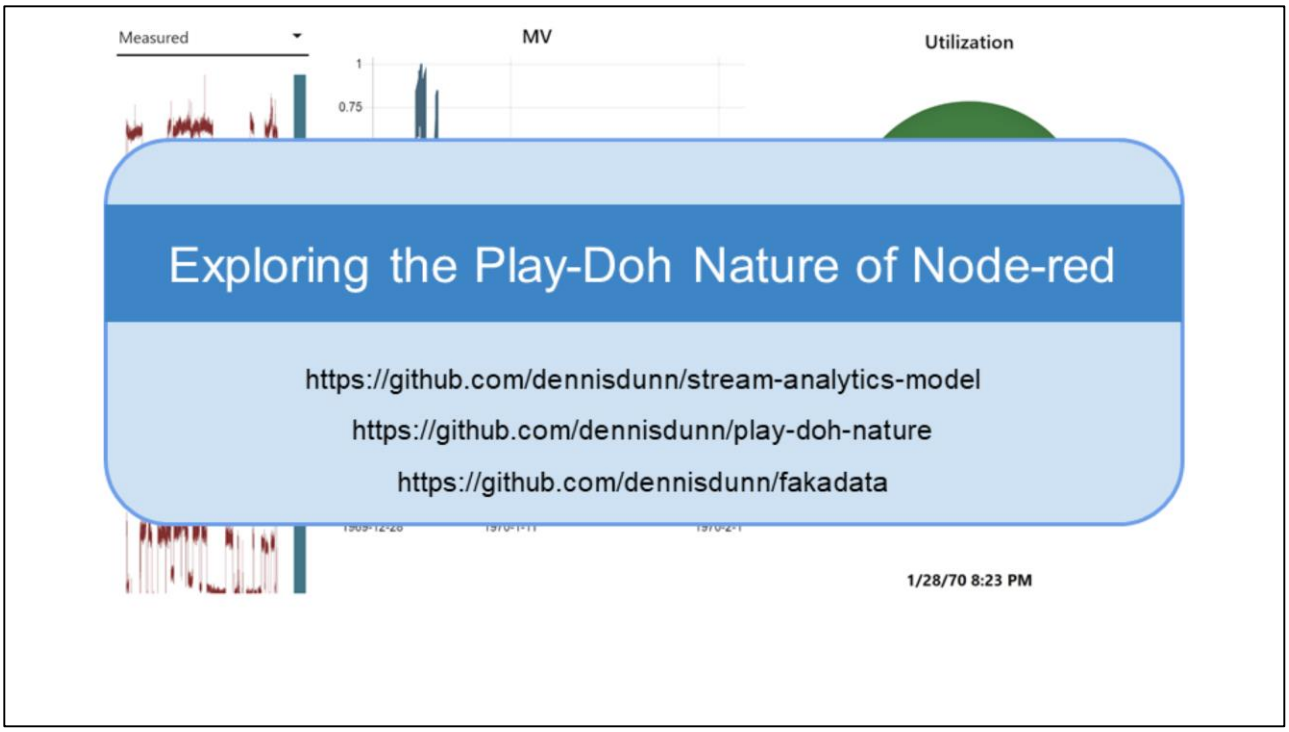


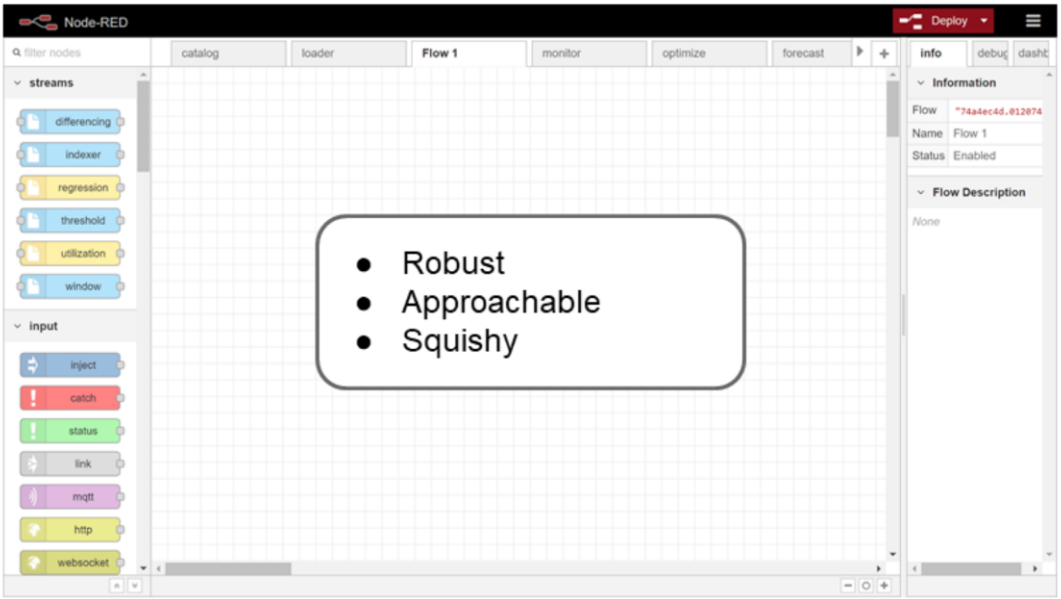
Hello everybody! Good to see you!

As programmers we write software to solve problems in many areas of human endeavour and those problems can be very complex.

Sometimes we can find ourselves wanting to dial-down the complexity so that we can get a better understanding of whatever the big complex thing is.

To do that we build a model; the material used to build the model should have the Play-Doh nature. Like the child's modeling clay, it should be:

- Robust
- Approachable
- Squishy



-the play-doh nature-

Node-red has the Play-Doh nature.

Node-red is a flow-based programming environment for the internet-of-things.

It is **robust.** I've used it to expose a web API for my rover bot.

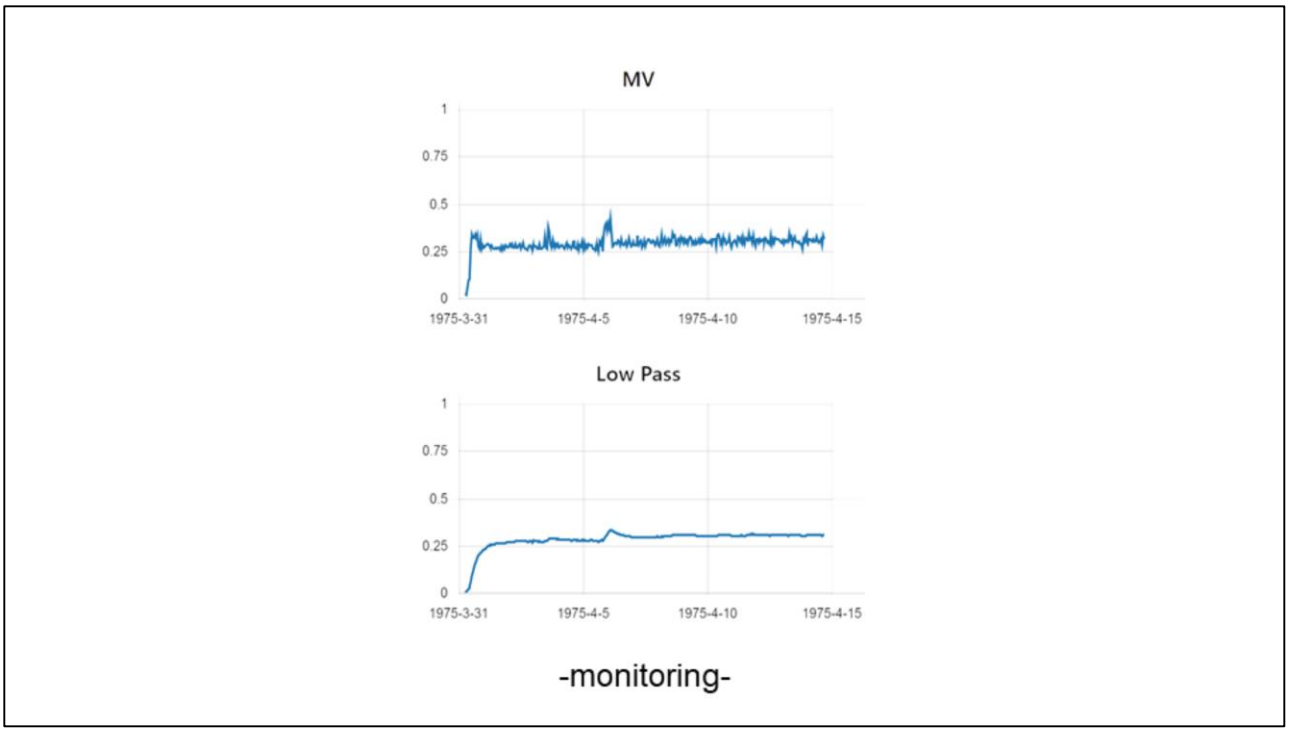
It is **approachable**. It uses javascript under the hood.

Finally, node-red has the all-important **squishy** property. If you need specialized functionality you can build it.

We are going to take a look at how node-red helps understand complexity while building we build a model.

My current client is in the process industries space and

writes software to monitor and analyze those processes. There are industry standards and protocols for collecting and transferring the data from these factories for analysis. Traditionally the analysis has been batch-oriented but with the industry moving to a digital-twin model this batch processing is exacerbating the lag between data collection, data analysis, and operations. You want to know about problems **before** your multi-million dollar plant goes down because of a burned out motor.



A digital twin is a digital replica of a physical asset or process. When Tony Stark waves his hands through a floating blueprint he is manipulating a digital twin.

A digital twin provides three main functions:

1. Monitoring
    a. What is happening now?
2. Diagnostics
    a. What's broken?
3. Forecasting
    a. What will happen in the future?

The model we'll be looking at is a digital twin of a pump like you might find in a petroleum refinery or a chemical

plant. These pumps have sensors that measure things like RPM and vibration and present time stamped measurements to be analyzed.

Let’s take a closer look at the **monitoring** section. This is where we would apply different digital-signal-processing algorithms to the stream of signal data and compare the processed stream to the raw stream.

One class of dsp algorithms are the **decomposition functions**.

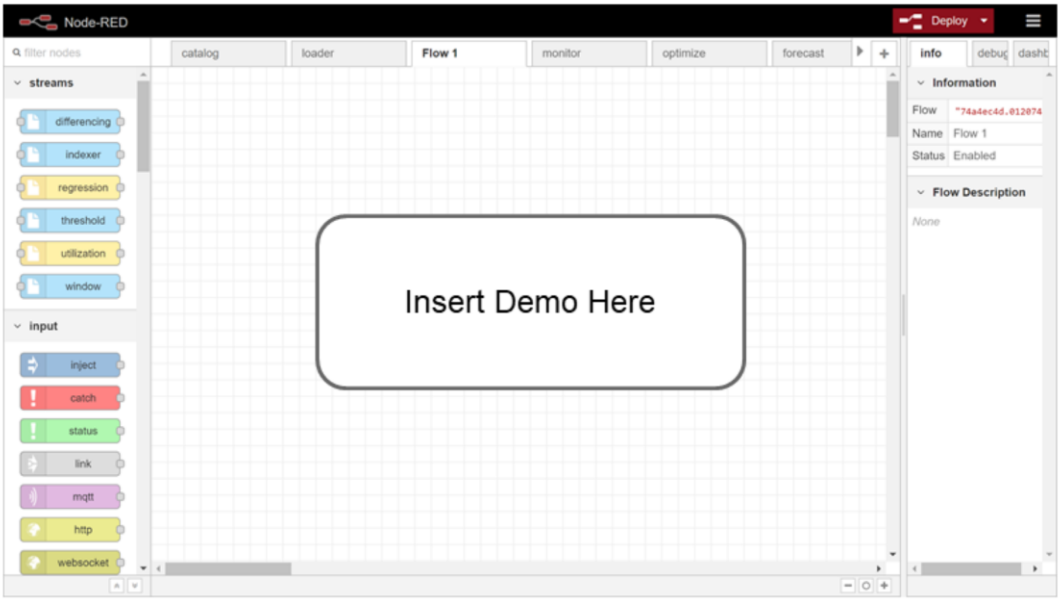
A digital signal can be decomposed into

- **Trend**, **seasonal**, and **residual** components

A low pass filter removes the seasonal and residual components leaving the trend.

A differencing filter removes the trend component and leaves the seasonal and residuals.

Enough jibber jabber. Let’s Demo!



-monitoring-

Monitoring - Demo Time

In node-red these tabs are called *flows* and are the basic unit of abstraction. Messages travel from node to node over the interconnecting *wires*, each node will do something with each message and pass it along to one or more downstream neighbors.

That bright blue blob on the left is a *link* node. Links are the mechanism by which we get messages from one flow to another. This one happens to be an input link, it is accepting messages from another flow.

The message passes down the wire to this *graph* node which is provided by the node-red-dashboard package. In

addition to line charts the node can be configured to display bar charts or pie charts.

This particular configuration is using the messages timestamp property for the x-value and the messages payload property for the y-value.

The next node we'll look at is this one labeled "low pass." Notice that the link node passed the message to the mv node as well as this low pass node.

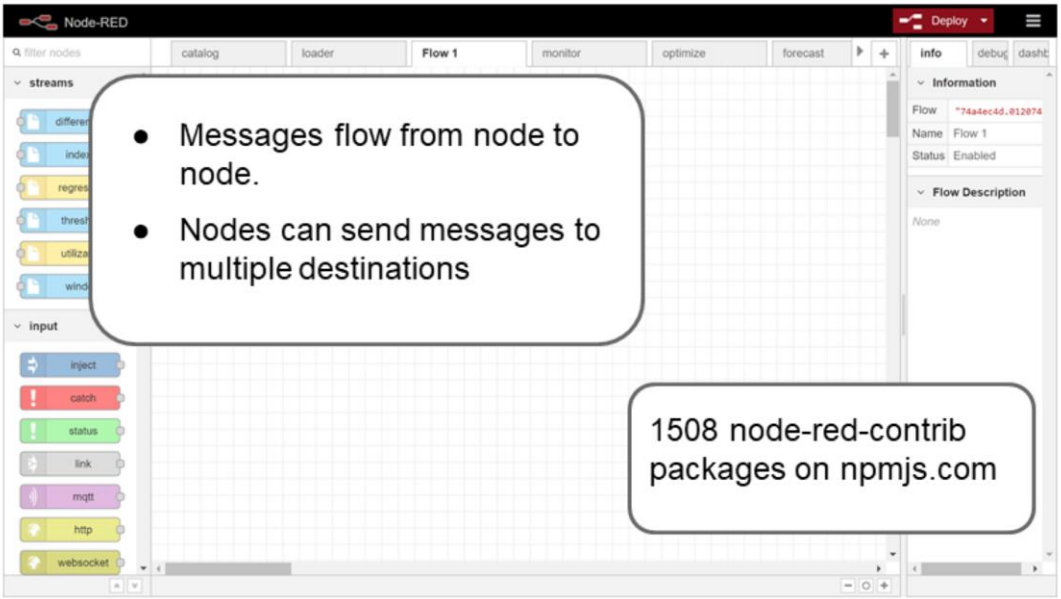
Node-red has multiple ways of sending messages to more than one downstream node. Messages can be duplicated like we see here or a message can be sent down different paths based on the value of a property on the message. Sometimes you might want to send the original node down one path and a modified node down another path.

This low pass filter comes from the node-red-node-smooth package and can be configured to apply different smoothing algorithms to the inputs. That is really handy for this model because it allows us to play with the data and we can see the changes on the charts.

Finally the low pass node sends the message to another chart node. This one is configured the same way as the mv chart and allows us to compare the measured value to the filtered value.

One of the neat things about the chart node … one of the

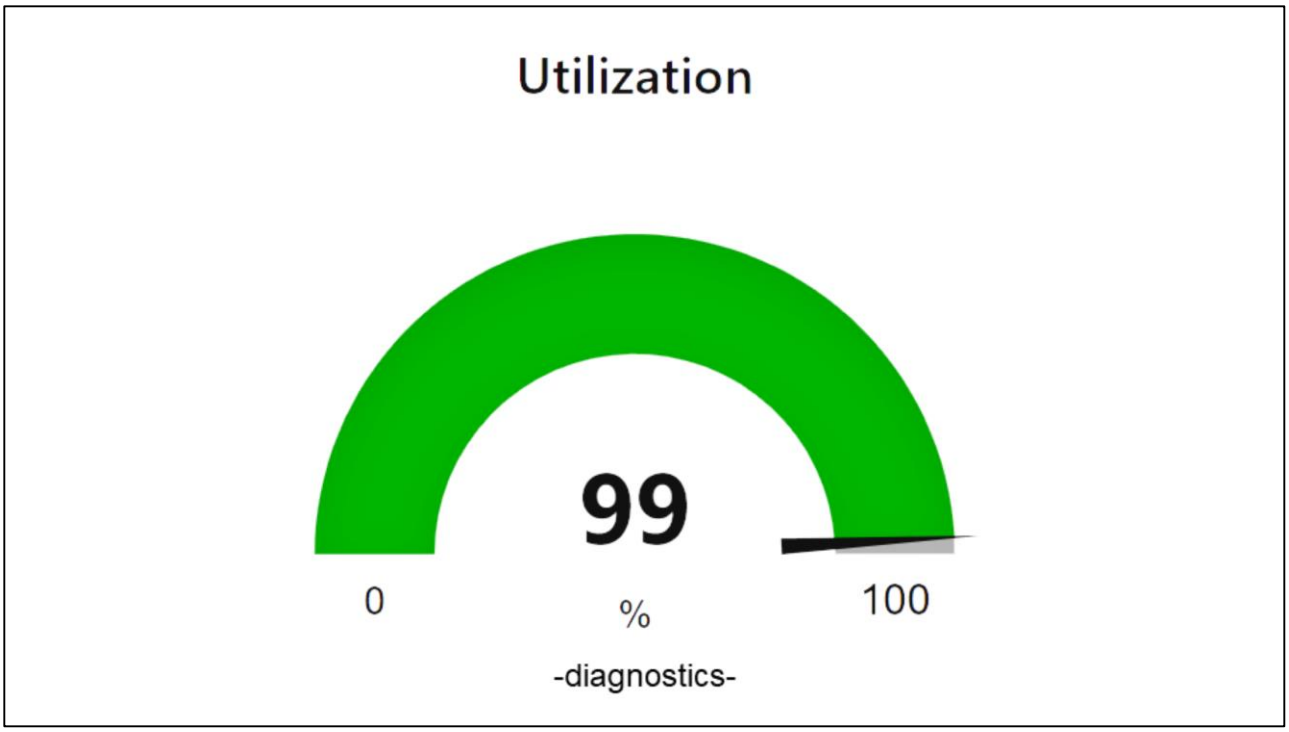
many neat things … is that you can send two data streams to a single chart and the chart will plot both streams.



-monitoring-

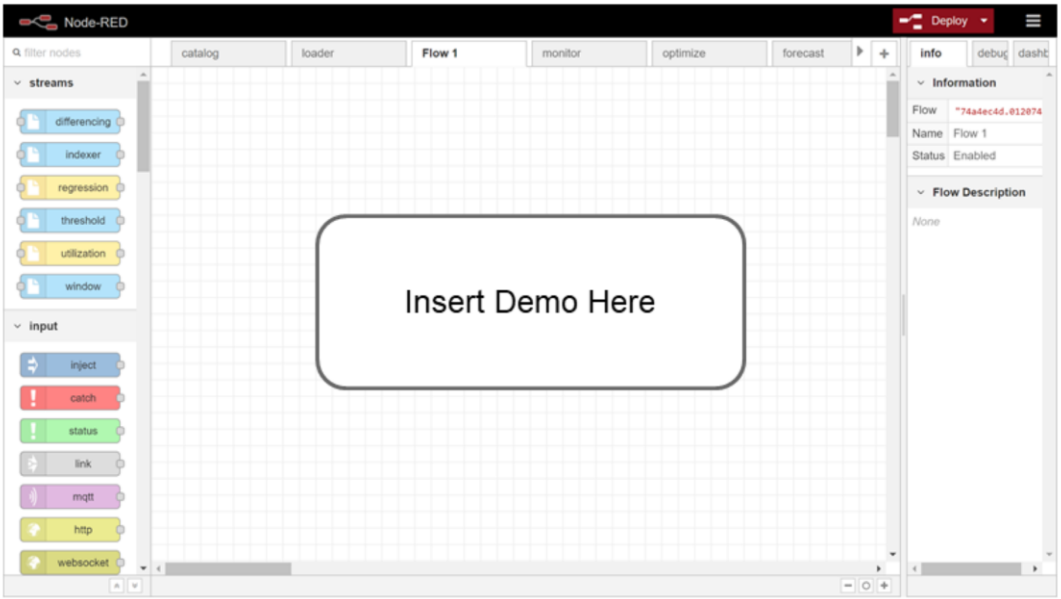
Monitoring - Takeaways

- Flow from left-to-right
- Messages have id, timestamp, topic, payload
- Nodes manipulate the messages as they pass through the node
- Messages can be sent down multiple paths
- There are 1500 packages in the npm registry that have the node-red-contrib-* prefix
- The entire functionality of this part of the model was built with out-of-the-box nodes and a couple of packages from the registry



The **diagnostics** section consists of a gauge which displays **utilization** which is the proportion of the time that the pump is in operation. If it's been on for 8 out of the last 10 days then the utilization is 80%.

Some manufacturers of these pumps provide signals from the pump which tell you the operational status of the pump but this one doesn't so we'll have to make do. For our purposes we'll count the machine as being operational if the vibration signal is over 0.1.



-diagnostics-

Diagnostics - Demo Time

Here we see the low pass filter again and another dashboard node which renders the gauge.

The interesting node is that bright yellow one on the right. That is where we calculate the utilization from the stream of velocity samples entering the flow through those link nodes at the upper left.

That yellow utilization node is an example of a *user-defined node*, it is defined by two files - an html file and a javascript file. The html provides the user interface for the node configuration and the javascript implements the functionality of the node.

We haven’t talked about configuration have we? Now seems like a good time for a short detour.

This is the configuration panel for the utilization node. This is where we can change that 0.1 we picked earlier to another value.

In the red box are the edit fields for the properties of the node. Like any proper web app the form has validation, default values, and placeholder text.

The purple box is the node help which tells the user what the node does. This node has an example of a fairly common special behavior - if the message has a property called “reset” then the node will clear any internal data structures and start over.

User-defined nodes have an html file and a javascript file.

This html fragment defines the form fields and the help text. If you’ve spent any time at all with html then the divs and inputs won’t be very exciting.This is the other part of the html file, notice that the script tag just has a boring old javascript MIME type.

This fragment sets up visuals of the node like color and icon as well as default values for the edit fields.

Node-red provides two validators, one for numbers and the other is for strings and uses regular expressions.

If you have a node that has some very specific editing needs you can define your own validators and hook into the edit life cycle with callbacks.

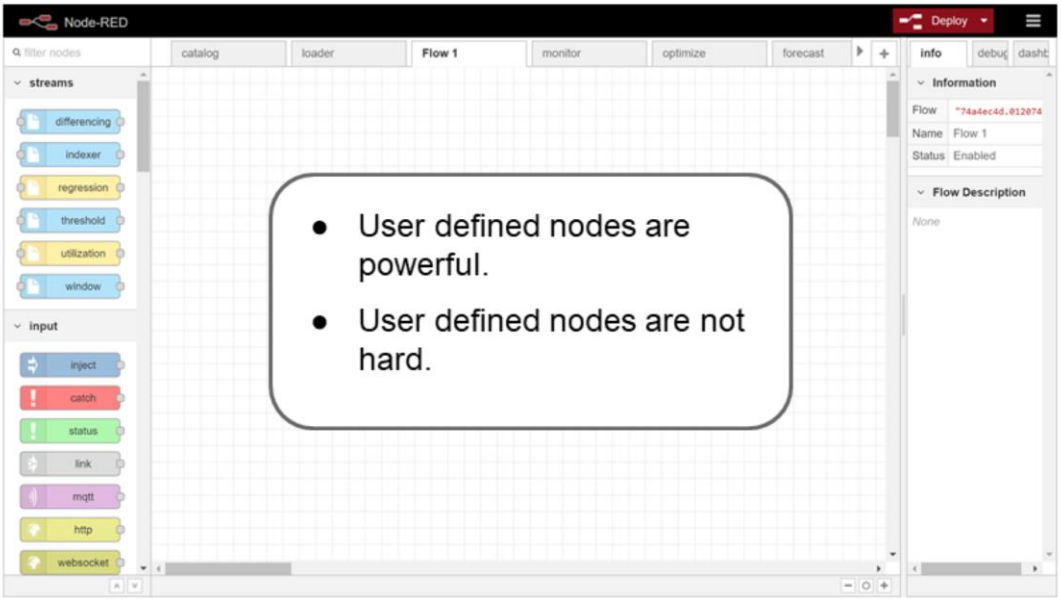
At this point the configuration editor is defined but what is this node going to actually **do**?

For that we need ...

… a little more javascript. I'm beginning to think of javascript as the duct tape of the internet.

The first thing of interest here is the **Utilization** function, this is a factory that creates the node and sets up an event listener for the **input** event. This factory gets registered with the run-time in that last line.

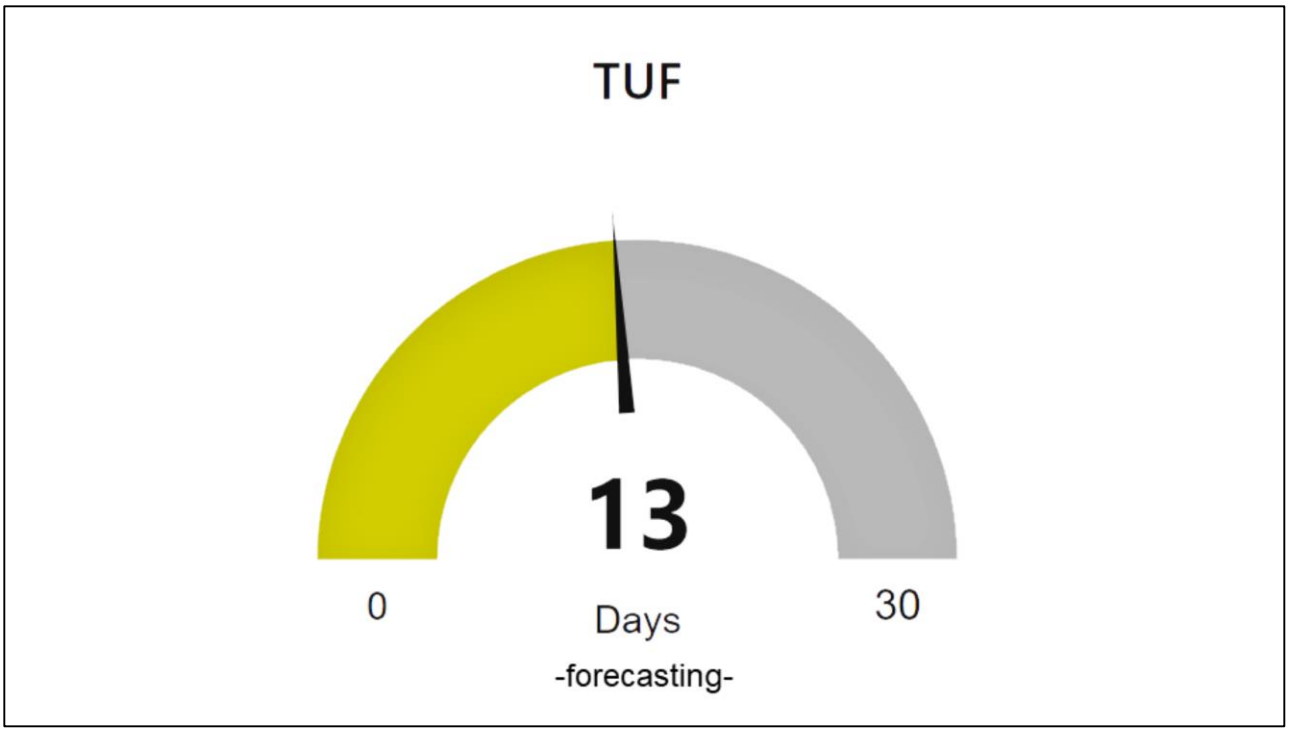
The other thing of interest is the event handler - this is where the node does its work. In this case it keeps track of the sum of the time differences between messages and the sum of the time differences where the value of the message was greater than the configured threshold. With each message that comes in to the node it updates the sums, calculates **on** time divided by **total** time, and sends the message downstream.



-diagnostics-

Diagnostics - Takeaways

- You can define your own nodes with their own configuration UIs
- Useful when you need multiple instances with different configurations



In this section we want to forecast the **time-until-failure** for the pump. Knowing when something is going to break allows you to plan ahead and replace the asset on *your* schedule and minimize downtime, costs, and lost revenue.

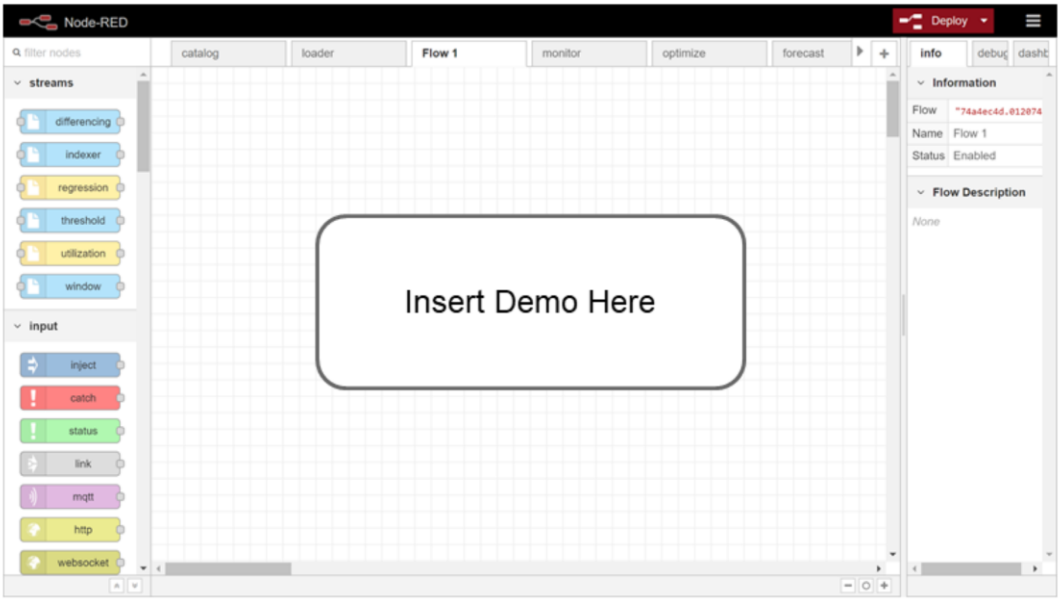
Just like in the utilization kpi we'll pick an arbitrary threshold to indicate failure. Let's define failure as the point where the vibration signal is greater than 0.75. We want to know when the red trend line crosses over the threshold. The tricky part is that we want to know before it happens, preferably days before it happens.

We're going to do that by using a regression model, setting *v* to 0.75 and then solving for *t*. That should tell us when

the pump will fail. When the vibration isn’t getting any worse the blue line won’t cross the threshold for a quite while.

When the vibration starts to change more dramatically you can see how the green line will cross the threshold a lot sooner.

I know that there is someone in the audience thinking “That looks like a parabola. Just do a 2nd order regression and be done with it!” You are absolutely correct. However, this is fake-fake data meant to help visualize the kpi calculations. We’ll look at some real-fake data later.



-forecasting-

Forecasting - Demo Time

The three yellow nodes on the left are doing some processing of the data stream to get it ready for the next node. The first one should be familiar, it's the low pass node we saw in earlier flows. It's followed by two nodes that group messages together. Before I understood how the **batch** and **join** nodes worked, I wrote a user-defined node that provides the same functionality. Not only was it easy to do it was ultimately unnecessary.

The batch node adds a property to each message as it passes through the node - that prop indicates the messages position in a specific group of messages. The

join node then looks at the property and bundles related messages into an array and sends the array downstream. The batch node allows you to configure the window size and if you need a rolling window you can configure the size of the overlap.

Just like the smooth operator in the monitoring flow and the threshold value in the diagnostics flow, we can manipulate the batch node configuration and observe how those changes affect the outcome.

The yellow regression node is a user-defined node that makes use of an npm package. Packages are made available through the global context by *requiring* the package in the node-red settings.js file.

Here you can see that we set four properties on the global context - one string and the results of three calls to require(). These properties are available to all user-defined nodes as well as function nodes.

This is the code for the regression node. The first thing that happens in the event handler is we grab the regression package from the global context. The next bit of interest is the call to map() to coerce the data into the form that the regression package needs. Recall that the

messages have timestamp and payload properties on the message object, the regression package wants a list of lists.

Don’t get fooled by the **order** property of that options object. It is effectively ignored because we’re calling the .linear() function. The regression package also has .exponential(), .logarithmic(), .power(), and .polynomial() regression models.

After all of that smoothing and regression-ing we can finally do some forecasting in the **forecast** node.

The forecast node is implemented as a node-red **function** node. A function node is like the event handler portion of a user-defined node.

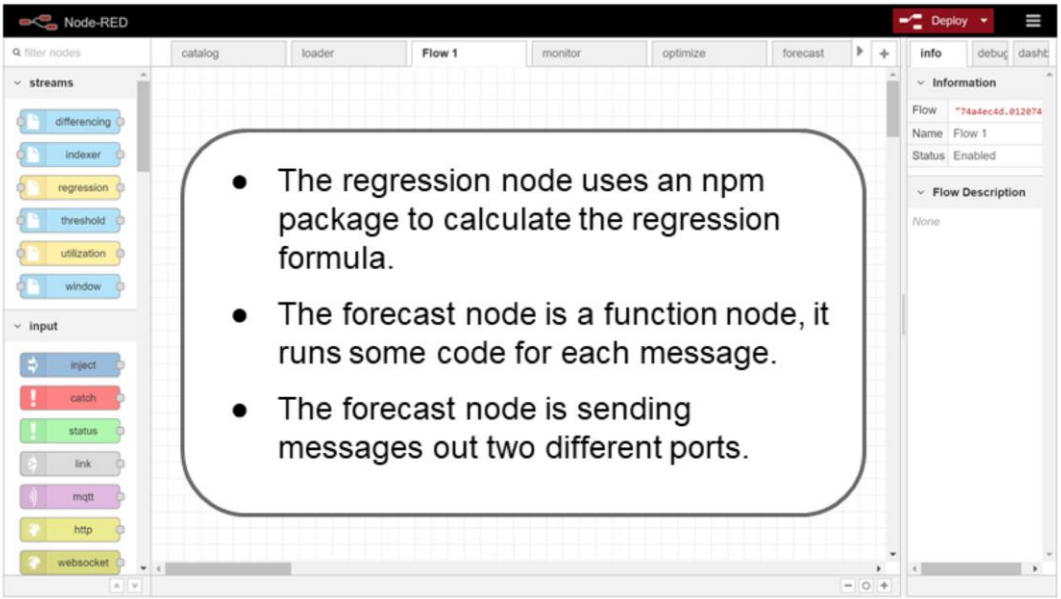
There are really only two things going on in this node that are of any interest. First is that third line where we’re using the coefficients from the regression model to figure out at what time the regression crosses the 0.75 threshold.

The next thing of interest is the return statement.

When you set the properties of a function node you can define the number of output ports the node will have. So far, all of the user-defined nodes that we’ve looked at

return a single message object but here we see that the function is returning an array of messages. Each message in the array will be sent out of a different port. The first message has been formatted for the gauge widget and the second output is formatted for displaying the projected time-of-failure in a text widget.

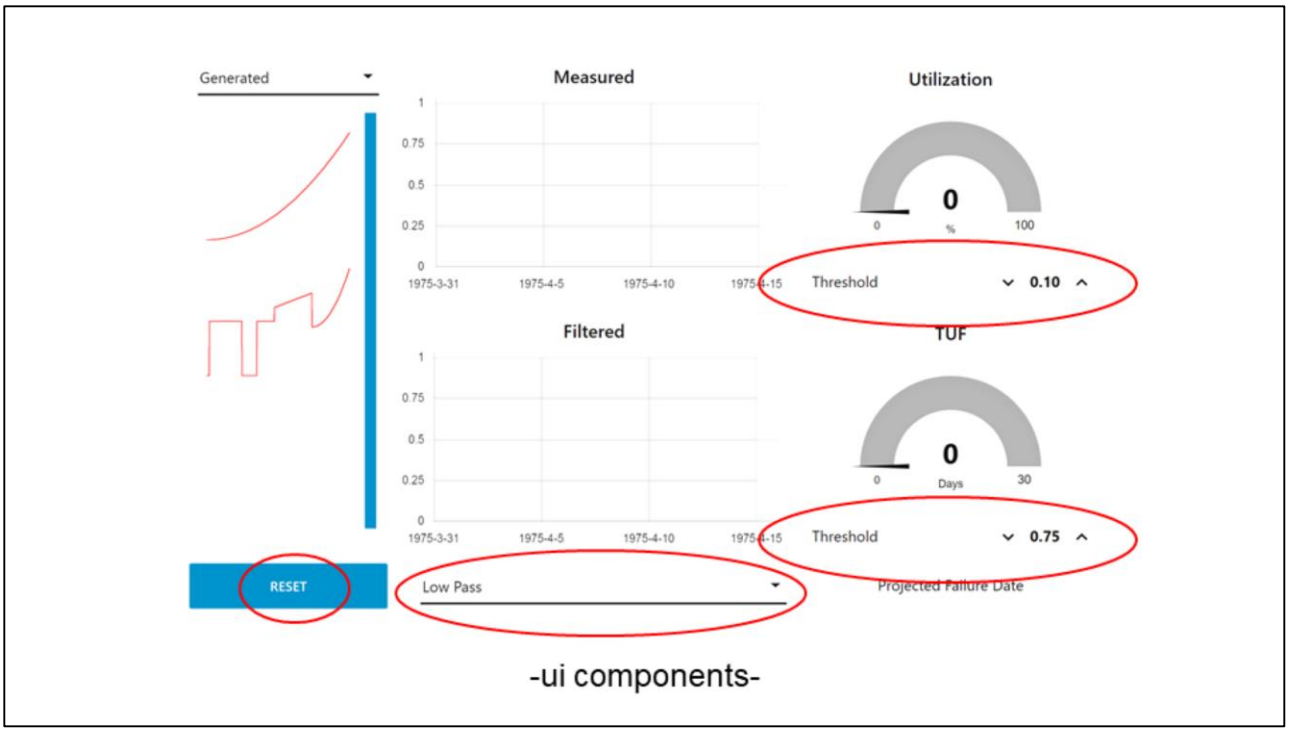
It’s like a function that has TWO return values! Pretty neat, huh?



-forecasting-

Forecasting - Takeaways

- User-defined nodes - we've already covered them.
- The regression node uses an npm package to calculate the regression model
- The forecast node is a **function** node, it runs some code for each message
- The forecast node is sending messages out two different ports



-ui components-

I've mentioned a few times that one of the reasons to build a model is to have the ability to change the configuration of the model. It's the difference between these models of the **SR-71 Blackbird**. The one on the left flies and the one on the right doesn't.

It's the difference between a static and a dynamic model.

At this point our digital twin model is quasi-dynamic; we can change settings like the utilization threshold through the configuration panel but then we have to deploy the change to the runtime environment.

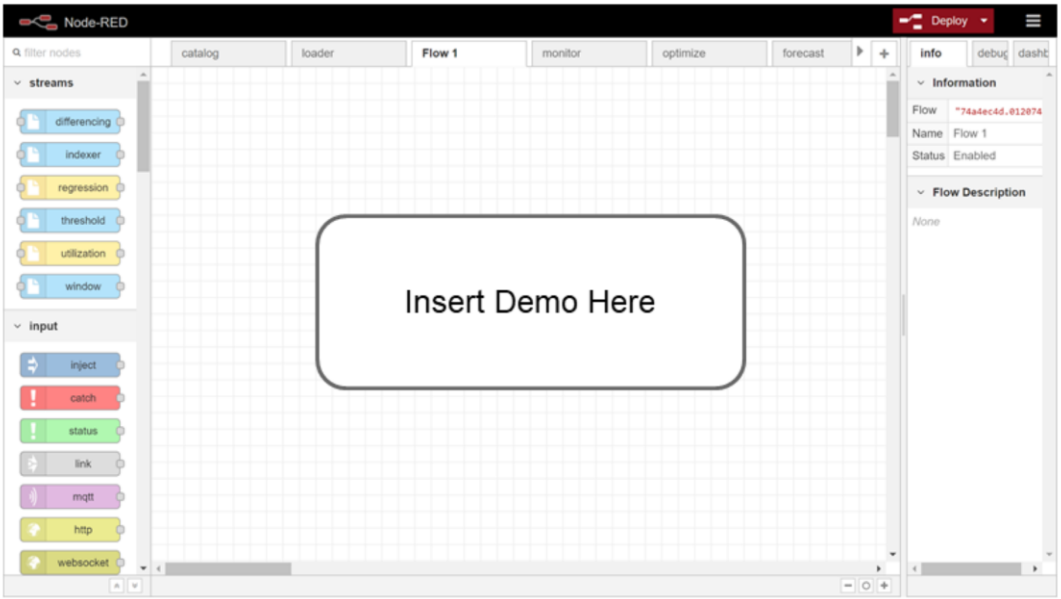
In addition to its data visualization nodes, the dashboard package also has data input nodes.These widgets allow us to change the behavior of the system by injecting

messages into the flow.

The numeric inputs on the right change the values of the utilization and failure thresholds

… while the dropdown in the center selects different filters

… and the button on the left sets everything back to sane values.



-ui components-

UI Components - Demo Time

The numeric inputs send a message each time you click on the up or down arrows, the payload of the message is the new value of the control.

That message is sent to a couple of downstream neighbors...

… a link node and a function node. The link provides a mechanism for alerting other flows that the value of the threshold has changed.

The function node takes the new value and stores it in the global context.

Each node has three contexts available to it at the

- **Global**, **flow**, and **node** level

The utilization node was refactored to read the threshold from the global context and to use that value in the calculation of the utilization KPI.

For selecting the filter we have a **dropdown** node. Like the numeric input node, the dropdown node sends a message downstream each time its value changes.

In this filter selector flow, the downstream neighbor of the dropdown is a function node. The selected filter is being saved to the flow context where we'll use it to route data messages to the correct filter node.

The smooth node which provides the different filtering functions doesn't have the ability to set the filter type on the fly so we need to use a different mechanism to provide multiple filters.

The switch node labeled **route** is an example of sending a message out different ports based on a property. In this case, we route the message according to the *filter* property of the flow context.

The smooth node provides three different filters as well as a few statistics about the data stream. Here you can see that we've used three instances of the smooth node, each with different settings.

Data messages come in through the link and get passed to the switch node when routes them to one of the filters. The filtered result is then made available to the other flows through the output link.

You probably noticed that we had four options for on the dropdown node but only three types of filter in the smooth node.

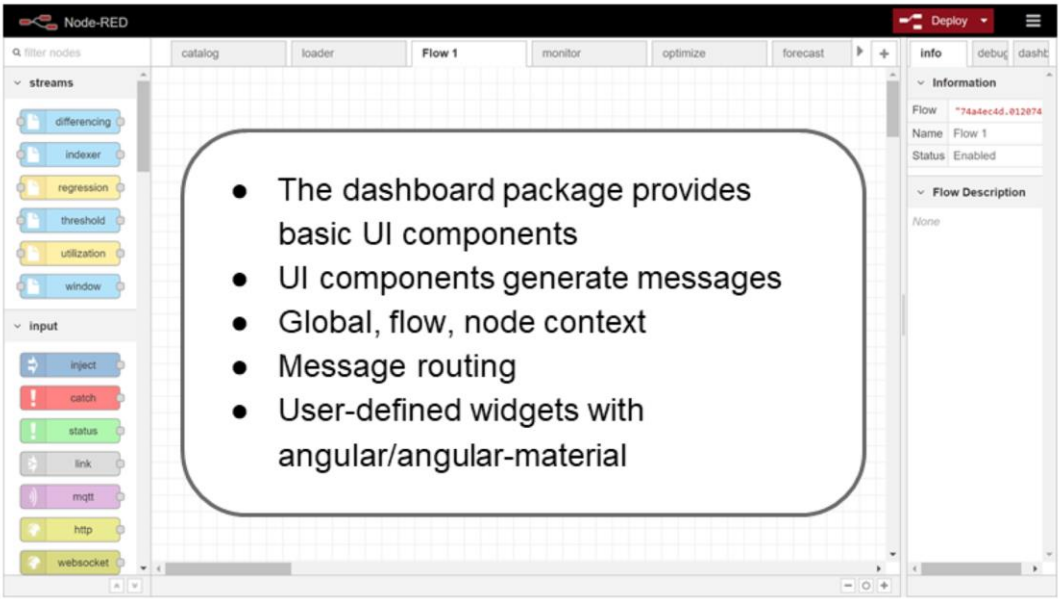
The fourth filter is defined in a function node. You've seen the batch/join sequence before, it is creating an array of messages. The median filter simply takes the array, sorts it, and returns the message in the middle.

Since we're on the subject of user interfaces, what happens if the dashboard package doesn't have the widget you need? You build one, of course.

I needed a way of displaying a bunch of images which would emit a message when clicked. That scrollable list of buttons on the left.

It turns out that you can use angular and angular-material directives in the template node which is part of the dashboard package.

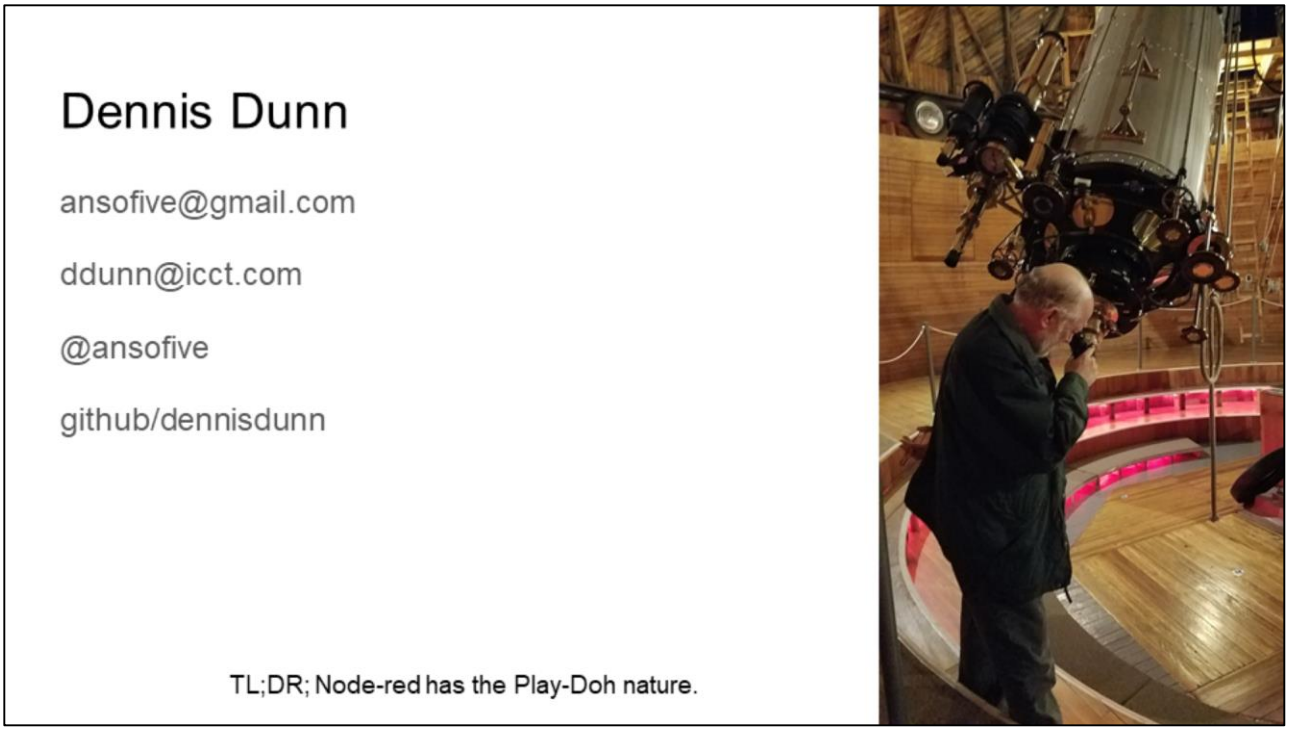
Now *that* is squishy!



-ui components-

UI Components - Takeaways

- The dashboard package provides basic UI components
- UI components generate messages
- Global, flow, node context
- Message routing
- User-defined widgets with angular



My name is Dennis Dunn and I'd like to thank you for coming. Here is some contact info and resources for this talk. I would love to answer any questions I can or we could just try the whole conversation thing.